N91-20698

ART-Ada: An Ada-Based Expert System Tool

S. Daniel Lee and Bradley P. Allen

Inference Corporation
550 N. Continental Blvd.
El Segundo, CA 90245

Abstract

The Department of Defense mandate to standardize on Ada as the language for software systems development has resulted in increased interest in making expert systems technology readily available in Ada environments. NASA's Space Station Freedom is an example of the large Ada software development projects that will require expert systems in the 1990's. Another large-scale application that can benefit from Ada-based expert system tool technology is the Pilot's Associate (PA) expert system project for military combat aircraft. This paper describes ART-Ada, an Ada-based expert system tool. ART-Ada allows applications of a C-based expert system tool called ART-IM to be deployed in various Ada environments. ART-Ada is being used to implement several prototype expert systems for NASA's Space Station Freedom Program and the U.S. Air Force.

1. Introduction

The Department of Defense mandate to standardize on Ada as the language for software systems development has resulted in increased interest from developers of large-scale Ada systems in making expert systems technology readily available in Ada environments. Two examples of Ada applications that can benefit from the use of expert systems are monitoring and control systems and decision support systems. Monitoring and control systems demand real-time performance, small execution images, tight integration with other applications, and predictable demands on processor resources; decision support systems have somewhat less stringent requirements.

An example project that exhibits the need for both of these types of systems is NASA's Space Station Freedom. Monitoring and control systems that will perform fault detection, isolation and reconfiguration for various on-board systems are expected to be developed and deployed on the station either in its initial operating configuration or as the station evolves; decision support systems that will provide assistance in activities such as crew-time scheduling and failure mode analysis are also under consideration. These systems will be expected to run reliably on a standard data processor, currently envisioned to be an 80386-based workstation. The Station is typical of the large Ada software development projects that will require expert systems in the 1990's.

Another large-scale application that can benefit from Ada-based expert system tool technology is the Pilot's Associate (PA) expert system project for military combat aircraft [3]. Funded by the Defense Advanced Research Projects Agency (DARPA) as part of its Strategic Computing Program, the PA project attempts to automate the cockpit of military combat aircraft using Artificial Intelligence (AI) techniques. A Lisp-based expert system tool, ART (Automated Reasoning Tool), was used to implement one of the two prototypes built during Phase I. An Ada-based expert system tool can provide a migration path to deploy the prototype on an on-board computer because Ada cross-compilers are readily available to run Ada programs on most embedded processors used for avionics.

Inference has been involved with Ada-based expert systems research since 1986. Initial work centered around a specification for an Ada-based expert system tool [4]. In 1988, the ART-Ada Design Project was initiated to design and implement an Ada-based expert system tool [6], [10], [11]. At the end of 1989, ART-Ada was released to beta sites as ART-Ada 2.0 Beta on the VAX/VMS and Sun/Unix platforms [7]. In 1990, eight beta sites, four NASA sites and four Air Force sites, will be evaluating ART-Ada 2.0 for eight months by developing expert systems and deploying them in Ada environments. The objectives of the ART-Ada Design Project were two fold:

1. to determine the feasibility of providing a hybrid expert system tool such as ART in Ada, and

2. to develop a strategy for Ada integration and deployment of such a tool.

Both of these objectives were met successfully when ART-Ada 2.0 beta was released to the beta sites.

Inference Corporation developed an expert system tool called ART (Automated Reasoning Tool) that has been commercially available for several years [5]. ART is written in Common Lisp and it supports various reasoning facilities such as rules, objects, truth maintenance, hypothetical reasoning and object-oriented programming. In 1988, Inference introduced another expert system tool called ART-IM (Automated Reasoning Tool for Information Management), which is also commercially available [8]. ART-IM is written in C and it supports a major subset of ART's reasoning facilities including rules, objects, truth maintenance and object-oriented programming. ART-IM supports deployment of applications in C using a C deployment compiler that converts an application into C data structure definitions in the form of either C source code or object code. ART-IM's interactive development environment includes a graphical user interface that allows browsing and debugging of the knowledge base and an integrated editor that offers incremental compilation. ART-IM is available for MVS, VMS, Unix, MS-DOS, and OS/2 environments.

Our approach in designing an Ada-based expert system tool was to use the architecture of proven expert system tools: ART and ART-IM. Both ART and ART-IM have been successfully used to develop many applications which are in daily use today [1], [12], [13]. ART-IM was selected as a baseline system because C is much closer to Ada. While ART-IM's inference engine was reimplemented in Ada, ART-IM's front-end (its parser/analyzer and graphical user interface) was reused as the ART-Ada development environment. The ART-IM kernel was enhanced to generate Ada source code that would be used to initialize Ada data structures equivalent to ART-IM's internal C data structures, and also to interface with user-written Ada code. This approach allows the user to take full advantage of the interactive development environment developed originally for ART-IM. Once the development is complete, the application is automatically converted to Ada source code. It is, then, compiled and linked with the Ada runtime kernel, which is an Ada-based inference engine.

2. Overall Architecture

ART-Ada is designed to be used by knowledge engineers who may not be familiar with Ada. With minimum knowledge about Ada, they can still develop a knowledge base in a high-level language whose syntax most resembles that of Common Lisp. When the knowledge base is completed, Ada source code can be generated automatically by simply "pressing a button".

When this automatically generated Ada code is compiled and linked with the Ada library of the ART-Ada runtime kernel, an Ada executable image is produced. ART-Ada also provides extensive capabilities for Ada integration so that the knowledge base can be embedded in an Ada environment. It would be best if the knowledge engineer developing the knowledge base works with an Ada programmer who serves as a system integrator. ART-Ada would be most useful for those who must deploy in Ada environments (because of the Ada mandate) expert system applications already developed using tools that do not support Ada deployment.

The overall architecture of ART-Ada is depicted in figure 2-1. The knowledge base is developed and debugged using an interactive user interface that supports three main features; a command loop similar to the Lisp eval loop, a graphical user interface for knowledge base browsing and debugging, and an integrated editor for incremental compilation of the knowledge base. Any user-written Ada code can be integrated into the knowledge base by either calling it from a rule or invoking it as a method for object-oriented programming.

Once the knowledge base is fully debugged, it can be automatically converted into an Ada package for deployment. The ART-Ada runtime kernel is an Ada library, which is in essence an Ada-based inference engine. An Ada executable image is produced when the machine-generated Ada code and any user-written Ada code, if any, are compiled and linked with the Ada library.

3. Knowledge Representation

ART-Ada's key feature is the integration of rule-based representation and object-based (frame-based) representation. It supports three different programming methodologies:

- Rule-based Programming -- Rules opportunistically react to changes in the surrounding database. Rules can fire (execute) in an order

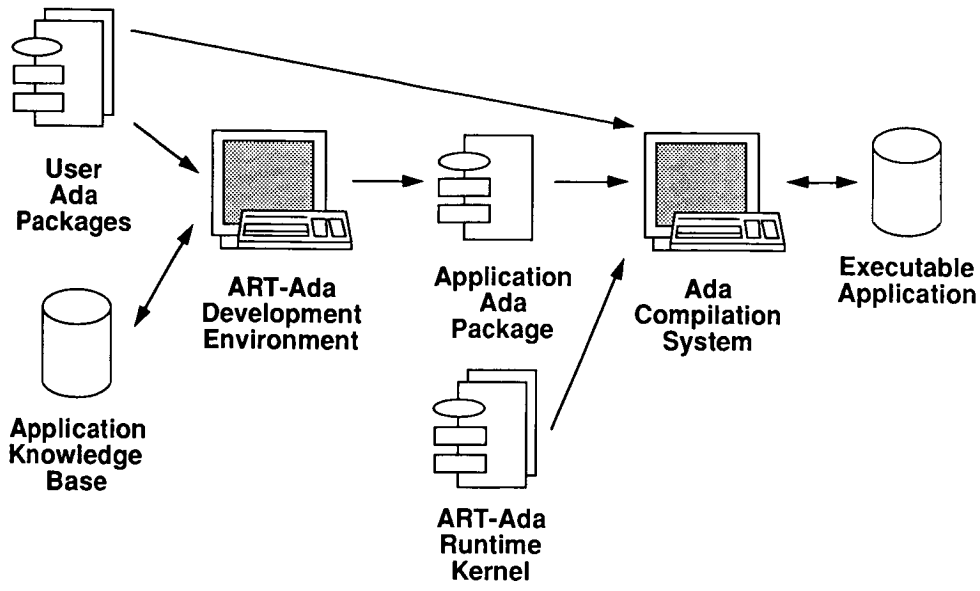

Figure 2-1: Overall Architecture of ART-Ada

based largely on the dynamic ordering of those changes. Rules cannot call other rules, and hence must communicate indirectly by making changes to the database which will, in turn, stimulate other rules.

- Object-Oriented Programming -- The fundamental unit of ART-Ada's object-oriented programming is the *object*, represented by a *schema*. Control is managed by sending *messages* to *objects* (schemas). The object reacts to the message by searching within itself for a *method* appropriate to that message. If an object does not have a method for the received message, it searches to see if it has inherited any appropriate methods from its parents. Once a method has been found, the object carries out the actions associated with the method.

- Procedural Programming -- ART-Ada's procedural language supports function calling, iteration (for, while) and conditionals (if, and, not). There are more than two hundred functions available in the procedural language.

ART-Ada's rule system is based on the optimized Rete pattern-matching algorithm [2]. Unlike OPS5, ART-Ada rules can pattern-match on objects called *schemas* as well as on lists called *facts*. *Facts* are similar to Lisp lists and do not support any inheritance. *Schemas* are similar to CLOS (Common Lisp Object System) objects; they are organized as attribute-value pairs and support *inheritance* through the *is-a* (subclass) and *instance-of* (member) relations. In the following example, *mammal* and *dog* are schemas while (animal-found dog) is a fact. *Mammal* is a class and *dog* is a subclass of the class *mammal*; they are linked with an *is-a* link. On the other hand, *fido* is a member of classes *dog* and *mammal*; it is linked to the class *dog* through an *instance-of* link. The significance of the relations *is-a* and *instance-of* is that the attribute-value pairs gets inherited either from a class to a subclass or from a class to a member. In the following example, *fido* will inherit attributes (eats meat), (socialization pack), (locomotion-mechanism run), and (instance-of mammal) from *dog*; it will also inherit (feeds-offspring milk) and (skin-covering hair) from *mammals*. As shown in the rule *determine-if-dog* that matches on both a schema pattern (schema ?animal (...)) and a fact pattern (classify-animal ?animal), the ART-Ada rules can match with schemas as well as facts. In order to optimize performance, ART-Ada uses two separate pattern matchers: one for schemas and one for facts.

```
(defschema mammal
   (feeds-offspring milk)
   (skin-covering hair))

(defschema dog
   (is-a mammal)
   (eats meat)
   (socialization pack)
   (locomotion-mechanism run))

(defschema fido
   (instance-of dog)
   (owned-by John))
```

```
(defrule determine-if-dog
   "Determine if subject is a dog."
   (classify-animal ?animal)
   (schema ?animal
      (is-a mammal)
      (socialization pack)
      (eats meat))
   =>
   (assert (schema ?animal
             (is-a dog)))
   (assert (animal-found dog)))
```

When an expert system deduces a conclusion (e.g. to diagnose faults in an electric circuit), it is often required to answer a question like "why?". This capability is called *explanation*. In ART-Ada, an explanation capability can be implemented using the *justification system*. When enabled, the justification system can provide a listing of the rules and data objects which were responsible for creating a particular fact or schema. By embedding features of the justification system in an application, the expert system can trace the steps leading to a particular conclusion. The justification system is also a powerful debugging tool when used during the development of an expert system. Should an application exhibit unexpected behavior during development, the programmer can exploit the features of the justification system to discover the source of the problem.

In the following example, if (classify-animal my-kangaroo) matches with a LHS pattern (classify-animal ?animal) where ?animal is a variable, and the rule fires to assert (schema my-kangaroo (is-a marsupial)), then we say that (classify-animal my-kangaroo) *justifies* (schema my-kangaroo (is-a marsupial)). In ART-Ada, consistency of the knowledge base is maintained by a justification-based truth maintenance system (JTMS) called Logical Dependencies. If *logical* is wrapped around (classify-animal ?animal), (schema my-kangaroo (is-a marsupial)) is not only *justified by* but also *logically dependent* on (classify-animal my-kangaroo); when (classify-animal my-kangaroo) is retracted from the knowledge base, (schema my-kangaroo (is-a marsupial)) is also retracted, and therefore consistency of the knowledge base is maintained automatically.

```
(defrule determine-if-marsupial
   "Determine if subject is marsupial."
   (logical (classify-animal ?animal))
   (schema ?animal
      (is-a mammal)
      (carries-offspring pouch))
   =>
   (assert (schema ?animal
             (is-a marsupial))))
```

In ART-Ada, object-oriented programming can be used with rule-based programming to take advantage of both paradigms. In the following example, the rule *print-out-object* is used to sent the *print* message to all objects that are instances of *object*. When an object *my-triangle* matches with the rule *print-out-object*, an inherited method *print-triangle* will be invoked. Methods can be defined either in ART-Ada's procedural language using *def-art-fun* which is similar to the Lisp defun, or directly in Ada using *def-user-fun* which will be discussed later.

```
;;; define objects

(defschema object
  (print print-unknown))

(defschema circle
  (is-a object)
  (print print-circle))

(defschema triangle
  (is-a object)
  (print print-triangle))

(defschema my-triangle
  (instance-of triangle)
  (position (1 2)))

;;; define a rule that sends a print message.

(defrule print-out-object
  (schema ?object
          (instance-of object)
          (position (?x ?y)))
  =>
  (send print ?object ?x ?y))
```

4. Knowledge Base Debugging

ART-Ada offers three main features in the user interface called the *Studio*:*

- a command loop,
- a graphical user interface, and
- an integrated editor.

ART-Ada's command loop is similar to the Lisp eval loop, in which user input is interpreted. More than two hundred functions are available in the command loop. Even Ada functions can be added to the command loop and called from the command loop.

*The Sun version supports only a command loop interface while the VAX/VMS version supports all three.

The Studio's interactive, menu-based graphical user interface provides immediate access to the knowledge base, and lets you monitor any aspect of program development or execution via an integrated network of menus and windows.

The Studio also provides a tightly integrated interface to the GNU Emacs full-screen editor. This interface facilitates the ART-Ada program development process by providing a number of powerful capabilities, such as incremental compilation of ART-Ada code.

The ART-Ada Studio can be used to do the following:

- Develop and execute an ART-Ada application.
- Browse the knowledge base -- to examine declarative (facts/schemas) knowledge, procedural (rules) knowledge, and runtime state, such as matches and activations.
- Debug the knowledge base -- by setting breakpoints in the programs and tracing their execution.
- Develop applications incrementally -- by editing the knowledge base to change facts or rules, or to modify program interactively.
- Generate Ada source code.

The ART-Ada/VMS Studio is based on DECwindows. The Studio is also implemented using other user interface standards (e.g. PM, OSF/Motif, ISPF) on other platforms.

5. Ada Integration

A major feature of ART-Ada is its ability to integrate expert systems technology with Ada. ART-Ada supports three types of Ada integration:

- *Ada call-out* refers to an ability to call Ada subprograms (procedures and functions) from the knowledge base (rules and methods).
- *Ada call-in* refers to an ability to call ART-Ada's public functions from Ada.
- *Ada call-back* is a special case of Ada call-in and refers to an ability to call ART-Ada's public functions from an Ada subprogram called from the knowledge base using Ada call-out.

Designers of expert systems will want to develop their own Ada code to provide user and system interfaces for their applications. There also may be a need to interface expert systems with other Ada applications (e.g. a signal processing application). A primary benefit of incorporating Ada code into the knowledge base is that Ada code will execute faster than similar code written in the ART-Ada procedural language. A consistent Ada call-in and call-out interface is provided for both development and deployment environments so that user-written Ada code runs without modification when it is deployed in Ada. In order to illustrate how an Ada subprogram is called from the knowledge base, let's consider the following rule:

```
(defrule distance-calculation-rule
  "calc distance between airfield and base"
  (schema ?airfield
    (instance-of airfield)
    (lat ?lat1)
    (lon ?lon1))
  (schema ?base
    (instance-of base)
    (lat ?lat2)
    (lon ?lon2))
  =>
  (bind ?distance
    ;; call an Ada function to calc distance
    (calculate-distance ?lat1 ?lon1
                        ?lat2 ?lon2))
  (assert
    (distance ?base ?airfield ?distance)))
```

The function, *calculate-distance*, can be implemented either in the ART-Ada procedural language or in Ada, but the Ada version would run faster. The ART-Ada construct *def-user-fun* specifies the interface between ART-Ada and Ada. It establishes an ART-Ada function name which calls out to the corresponding Ada subprogram, and it provides a description of data being passed. For example, *calculate-distance* can be specified as an Ada function as follows:

```
(def-user-fun calculate-distance
  :args ((lat1 :float)
         (lon1 :float)
         (lat2 :float)
         (lon2 :float))
  :returns :float
  :compiler :dec-ada)
```

This *def-user-fun* statement specifies that the ART-Ada function *calculate-distance* will call out to an Ada function CALCULATE_DISTANCE. There are four arguments of a type floating-point number being passed to Ada. The return value is also a floating-point number. It also specifies the default Ada compiler for the platform (i.e. DEC Ada). The corresponding Ada code should be declared in a package called *USER* and would look like:

```
-- ART is a public package of ART-Ada.
with ART;
-- USER is a package for user's Ada code.
package USER is

  function CALCULATE_DISTANCE
    (LAT1, LON1, LAT2, LON2 : ART.FLOAT_TYPE)
       return ART.FLOAT_TYPE;

end USER;
```

ART-Ada	Ada	Size
integer	INTEGER_TYPE	32 Bits
float	FLOAT_TYPE	64 Bits
boolean	BOOLEAN_TYPE	
string	STRING	
symbol	STRING	
art-object	ART_OBJECT	

Table 5-1: Data Types for Ada Call-in/Call-out

Ada data types supported for the call-in and call-out interfaces are: 32 bit integer (INTEGER_TYPE), 64 bit float (FLOAT_TYPE), boolean (BOOLEAN_TYPE), string and symbol (STRING), and an abstract data type for objects in ART-Ada (ART_OBJECT). Table 5-1 summarizes the mapping between ART-Ada and Ada data types.

6. Ada Code Generation

ART-Ada takes one or more ART-Ada source files as input and outputs Ada source files that represent a single Ada package. At any point after ART-Ada source files are loaded into ART-Ada and the knowledge base is initialized for execution, the Ada code generator may be invoked to generate Ada source code. An Ada package specification generated by ART-Ada for an example application called MY_EXPERT_SYSTEM is shown below:

```
-- generated automatically by ART-Ada
package MY_EXPERT_SYSTEM is

  -- initialize the application.
  procedure INIT;

end MY_EXPERT_SYSTEM;
```

A simple Ada main program that initializes and runs the application MY_EXPERT_SYSTEM is shown below. It is the simplest way to run an ART-Ada application in an Ada environment. It is possible, however, to embed it in a large Ada program. ART-Ada's public Ada packages, ART and SCHEMA, include a full set of Ada utilities to control and access procedurally the knowledge base from Ada. In OPS5, for example, it is hard to access working memory elements procedurally. In ART-Ada, Ada utilities are provided to access the knowledge base directly from Ada.

```
-- This is a main program written by the user.
-- ART is a public package of ART-Ada.
with ART, MY_EXPERT_SYSTEM;
procedure MAIN is
  TOTAL_RULES : ART.INTEGER_TYPE;
begin
  MY_EXPERT_SYSTEM.INIT;          -- initialize
  TOTAL_RULES := ART.A_RUN(-1);   -- run it.
end MAIN;
```

In addition to generating the Ada source code that initializes the knowledge base, a call-out interface module is generated as a separate procedure; it is a large case statement that contains all Ada subprograms called out to from ART-Ada. ART-Ada also generates a command file used to compile all Ada files generated by ART-Ada.

7. Ada Runtime Deployment

The steps needed to deploy an ART-Ada application in Ada are summarized below:

1. Develop and debug an application using ART-Ada's interactive development environment. If necessary, call out to Ada using the call-in/call-out interface.

2. Generate Ada code from ART-Ada using the Ada code generator. If the Ada compiler platform is different from the ART-Ada development platform, the generated Ada code can be moved to the platform on which the Ada compiler runs as long as the ART-Ada runtime kernel is available for that platform.

3. Compile the generated Ada code and user-written Ada code using either a self-targeted compiler or a cross-compiler into an appropriate Ada library of the ART-Ada runtime kernel.

4. Create an Ada executable image by linking an Ada main program.

5. Deploy the Ada executable image on a host computer or on a target system.

8. Future Work

According to a recent benchmark, ART-Ada does not perform as well as ART-IM. While immature Ada compilers also contribute to the poor performance, fundamental problems of the Ada language itself have been uncovered [9]. Some examples are:

- dynamic memory management,
- function pointers, and
- bit operators.

Among these, the overhead of dynamic memory management is the most serious problem. Due to the dynamic nature of expert systems, it is necessary to allocate memory dynamically at runtime in ART-Ada and ART-IM. The direct use of *new* and *unchecked_deallocation* is the only dynamic memory management method available in Ada. The problem with this method is that *new* incurs a fixed overhead associated with each call and it is called very frequently to allocate a relatively small block for an individual data structure. It results in a performance penalty in size and the slower execution speed. This is also aggravated by the poor implementation of *new* in the Ada compiler.

The existing Ada features, *new*, *unchecked_deallocation*, and *unchecked_conversion*, are too restrictive and totally inadequate for a complex system that requires efficient memory management. More flexible features (perhaps in addition to the existing ones) should be provided. This is particularly important in embedded system environments that impose a severe restriction on the memory size.

This issue and others were presented to several members of the Ada 9X Project in a meeting held in Washington, D.C. in March, 1990. We believe that they should be addressed by the Ada 9X standard. Unfortunately, the revised Ada language based on the Ada 9X will not be available until 1993 or later, which would be too late for the Space Station Freedom software development schedule.

Our current research effort is focused on improving the performance of ART-Ada by implementing ART-Ada's own memory manager using current technology. If it is not possible to implement it in Ada, we will implement it in another language (e.g. an assembly language). ART-Ada has an Ada code generator, which generates Ada code that relies on new and unchecked_deallocation. The current code generator would have to be redesigned to be compatible with the new memory manager.

Other Ada language issues such as function pointers, bit operators and portability and compiler problems encountered during the development of ART-Ada are discussed elsewhere [11], [9].

9. Acknowledgments

The authors wish to acknowledge the guidance and support of Chris Culbert and Bob Savely of NASA Johnson Space Center, Greg Swietek of NASA Headquarters, and Captain Mark Gersh of the U.S. Air Force. Mark Auburn, Don Pilipovich, Mike Stoler and Mark Wright of Inference Corporation contributed to the project.

References

1. Dzierzanowski, J.M. et. al. The Authorizer's Assistant: A Knowledge-based Credit Authorization System for American Express. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.

2. Forgy, C.L. "RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem". *Artificial Intelligence 19* (1982).

3. Hugh, D.A. "The Future of Flying". *AI Expert 3*, 1 (January 1988).

4. Inference Corporation. Ada-ART, Specification for an Ada-based State-of-the-Art Expert System Construction Capability. Inference Corporation, August, 1987.

5. Inference Corporation. *ART Version 3.2 Reference Manual.* Inference Corporation, 1988.

6. Inference Corporation. ART/Ada Design Project - Phase I, Final Report. Inference Corporation, March, 1989.

7. Inference Corporation. *ART-Ada/VMS 2.0 Beta Reference Manual.* Inference Corporation, 1989.

8. Inference Corporation. *ART-IM/VMS 2.0 Beta Reference Manual.* Inference Corporation, 1989.

9. Lee, S.D. Toward the Efficient Implementation of Expert Systems in Ada. Submitted to the TRI-Ada Conference, ACM, 1990.

10. Lee, S.D., Allen, B.P. Deploying Expert Systems in Ada. Proceedings of the TRI-Ada Conference, ACM, 1989.

11. Lee, S.D., Allen, B.P. ART-Ada Design Project - Phase II, Final Report. Inference Corporation, February, 1990.

12. Nakashima, Y, Baba, T. OHCS: Hydraulic Circuit Design Assistant. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.

13. O'Brien, J. et. al. The Ford Motor Company Direct Labor Management System. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.